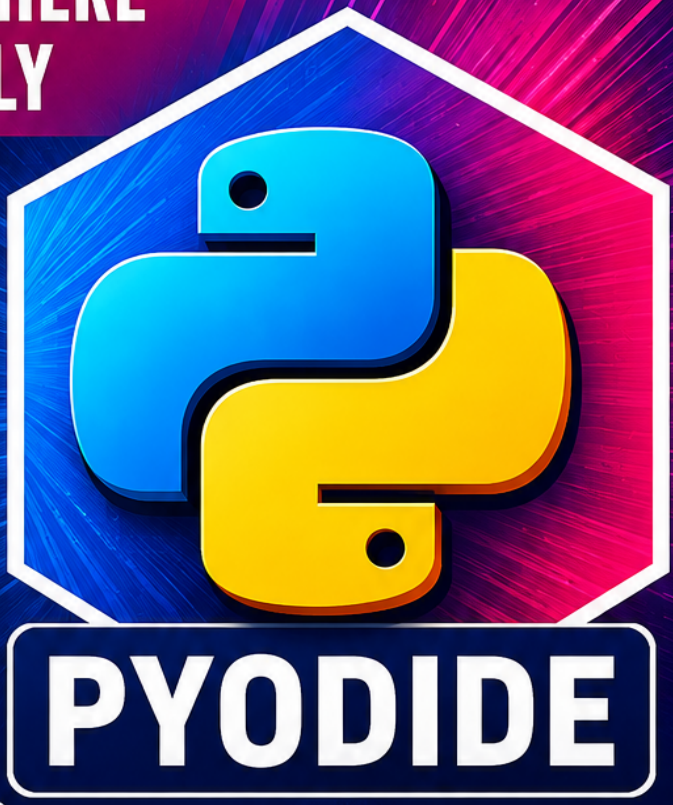
RUN REAL PYTHON IN THE BROWSER

PYODIDE IN ACTION

RUN PYTHON ANYWHERE WITH WEBASSEMBLY

A COMPLETE GUIDE FROM FIRST PRINCIPLES TO PRODUCTION USAGE

- RUN PYTHON IN THE BROWSER
- POWERED BY WEBASSEMBLY
- PRACTICAL EXAMPLES YOU CAN USE TODAY
- BUILD PRODUCTION-READY WEB AND EDGE APPLICATIONS



GEORGE WINSLOW

SOFTWARE ENGINEER AND ARCHITECT

Pyodide in Practice

Running Python in the Browser with WebAssembly: From Fundamentals to Production

Steve Publications

This book is available at https://leanpub.com/pyodideinpractice

This version was published on 2026-08-14

Contents

Running Python in the Browser with WebAssembly: From Fundamentals to Production

About this book: This comprehensive guide takes you from first principles to advanced production usage of Pyodide, the project that compiles CPython to WebAssembly so you can run real Python directly in web browsers and JavaScript environments. Whether you are a web developer wanting to add Python capabilities to your applications, a Python developer curious about client-side execution, or an architect evaluating serverless edge patterns with Pyodide, this book provides the technical depth, practical examples, and honest trade-off analysis you need to make informed decisions and build production-quality systems. Every chapter is grounded in current documentation and real-world usage patterns, with fully self-contained code examples you can adapt immediately.

Introduction: Python in the Browser

The Problem Pyodide Solves

Imagine building a web application where users upload CSV files and want to analyze them with pandas, generate visualizations with Matplotlib, or run machine learning models with scikit-learn, all without waiting for server round-trips, paying for backend compute, or exposing their data to your infrastructure. Or imagine an educational platform where students write and execute Python code directly in their browser, with a full scientific stack available and no local installation required. Or picture deploying Python-based logic to the network edge on platforms like Cloudflare Workers, bringing serverless simplicity to Python developers who prefer it over JavaScript.

These scenarios were largely impossible until WebAssembly matured enough to host real-world runtimes, and Pyodide emerged as the project that makes them practical today.

Before Pyodide, running Python in a browser meant one of several compromises. You could use Brython or Skulpt, which implement Python-like languages from scratch in JavaScript. These projects offer limited compatibility with standard Python libraries and cannot run C extensions at all. Alternatively, you could send code to a backend server for execution, introducing latency, infrastructure costs, and trust issues when running user-submitted code. For data science and interactive applications, neither option was satisfactory.

Pyodide solves this by taking the actual CPython interpreter (the reference implementation of Python 3 maintained by the Python Software Foundation) and compiling it to WebAssembly using Emscripten. The result is not a Python subset or a compatibility layer. It is real CPython running inside your browser, with access to most of the standard library and many popular third-party packages including NumPy, pandas, Matplotlib, SciPy, and scikit-learn.

This matters because it means existing Python code often works in Pyodide with minimal changes. Import statements that work on your development machine tend to work in the browser too. Libraries you already know behave familiarly. The learning curve for integrating Python into web applications

flattens considerably when developers can use tools and patterns they already understand.

What Is Pyodide? A Brief History

Pyodide is a Python distribution for browsers and Node.js based on WebAssembly and Emscripten [1]. It was created in 2018 by Michael Droettboom while working at Mozilla on the Iodide project, an experimental interactive scientific computing environment for the web that combined Jupyter notebook-style cells with JavaScript interactivity [2]. The name Pyodide comes from combining "Python" and "Iodide."

The key technical achievement was porting CPython to WebAssembly. Unlike projects that reimplement Python in another language, Pyodide takes the C source code of CPython and compiles it using Emscripten, a compilation toolchain that converts C and C++ code to WebAssembly. This approach preserves compatibility with C extension modules (the compiled C code that powers performance-critical parts of libraries like NumPy and pandas).

Since its inception, Pyodide has grown significantly:

- Early versions (2018–2020) established the core runtime and basic interoperability between Python and JavaScript.
- Version 0.17.0 (2020) added full asyncio support with a custom event loop implementation called WebLoop that integrates Python coroutines with the browser event loop [3].
- Version 0.18.0 introduced experimental Node.js support, ctypes functionality, and improved foreign function interface capabilities [4].
- Version 0.25.0 (January 2024) added support for synchronous HTTP via urllib3 and requests libraries, which had long been a community request [5].
- Version 0.28.0 (July 2025) focused on standardizing the Pyodide platform, including defining the Pyodide ABI and aligning with CPython's acceptance of Emscripten as a tier 3 target starting from Python 3.14 [6].
- Version 314.0.0 (June 2026) shipped with Python 3.14.2, reflecting the new versioning alignment with CPython releases [7].

The project is licensed under the Mozilla Public License Version 2.0 and is developed as an independent, community-driven open-source effort. It

receives support from organizations including Cloudflare, Quansight Labs, Anaconda, Symerio, CZI (Chan Zuckerberg Initiative), NSF, and Mozilla [8]. Infrastructure support comes from CircleCI for continuous integration, JsDelivr for content delivery, and ReadTheDocs for documentation hosting.

As of this writing, the latest stable release is version 314.0.4 (August 2026), running Python 3.14.2 [9]. The project maintains backward compatibility where practical but explicitly notes that some API aspects may change between versions, particularly around development-time tooling.

Real-World Use Cases and Who Should Care

Pyodide is not a toy technology. It powers real applications across several domains:

Interactive data science in the browser. Applications like Observable Plot's Python integration, Marimo notebooks running client-side, and various educational platforms use Pyodide to let users write and execute Python code with full scientific computing capabilities directly in their browser [10]. This eliminates server costs for computation and keeps user data local.

Client-side code execution and sandboxes. Platforms that need to run untrusted or user-submitted Python code (coding challenges, educational tools, configuration validators) can use Pyodide's WebAssembly sandbox as a safe execution environment. The browser's security model provides natural isolation without requiring container orchestration or separate compute infrastructure.

Edge computing with Python on Cloudflare Workers. Since April 2024, Cloudflare Workers has supported Python natively through Pyodide, allowing developers to write serverless edge functions in Python [11]. This brings the simplicity of Python's syntax and ecosystem to edge computing, with cold starts under one second thanks to WebAssembly snapshot technology introduced in late 2025 [12].

JavaScript framework integration. Modern web frameworks like React, Vue, Svelte, and Solid.js can integrate Pyodide to run Python logic alongside JavaScript UI code. This is particularly useful for applications where domain experts write business logic in Python while frontend developers build the interface in their preferred JavaScript framework.

Offline-capable Python tools. Because Pyodide runs entirely client-side after initial load, it enables offline Python development environments and tools that work without an internet connection once the runtime is cached. This is valuable for educational settings with limited connectivity or field work scenarios.

Who should read this book:

- **Web developers** who want to add Python capabilities to their browser applications without managing backend infrastructure.
- **Python developers** curious about running their code in browsers, whether for interactive tools, data visualization, or edge computing.
- **Full-stack architects** evaluating whether Pyodide fits into their technology stack, particularly for client-side computation or serverless patterns.
- **Educators and educational technologists** building interactive learning platforms where students execute Python code directly in the browser.
- **Data scientists** interested in sharing interactive analysis tools that run entirely on users' machines without requiring local Python installation.

The book assumes you are comfortable with basic web development concepts (HTML, CSS, JavaScript fundamentals, and how browsers work). You should also know basic Python syntax. No prior experience with WebAssembly or Pyodide is required; Chapter 1 covers the necessary WebAssembly background from scratch.

How This Book Is Organized

This book is structured as a progressive learning journey, building from foundational concepts through practical usage to advanced production patterns:

Part I: Foundations (Chapters 1–3): We start with the basics. Chapter 1 explains WebAssembly concepts you need to understand Pyodide without drowning in low-level details. Chapter 2 gets you running your first Pyodide applications through hands-on walkthroughs covering multiple setup approaches. Chapter 3 dives into Pyodide's architecture and runtime internals, giving you the mental model needed for debugging and optimization later.

Part II: Core Usage (Chapters 4–9): These chapters cover the essential skills for working with Pyodide effectively. Chapter 4 explores JavaScript–Python

interoperability in depth, including type conversions, proxies, and memory management. Chapter 5 covers package management with micropip and loadPackage. Chapter 6 explains virtual filesystems and I/O patterns. Chapter 7 tackles asynchronous programming with WebLoop and asyncio. Chapter 8 shows how to access browser APIs from Python. Chapter 9 demonstrates data science workflows with NumPy, pandas, and Matplotlib in the browser.

Part III: Advanced Patterns (Chapters 10–14): These chapters address production concerns. Chapter 10 covers networking, HTTP patterns, and running Pyodide on Cloudflare Workers for serverless edge computing. Chapter 11 provides performance optimization strategies including load time reduction and memory management. Chapter 12 equips you with debugging tools and techniques. Chapter 13 examines security considerations including Content Security Policy configuration and sandboxing properties. Chapter 14 guides you through deployment, bundler integration, framework patterns, and building custom packages.

Part IV: Limits and Future (Chapter 15): We conclude with an honest assessment of what Pyodide cannot do, when not to use it, and where the project is heading. Understanding limitations is as important as understanding capabilities when making architectural decisions.

Each chapter contains complete, runnable code examples with all necessary imports, setup code, and expected behavior explained. Examples are designed to be self-contained so you can copy them into your own projects and adapt them immediately. Where APIs or behaviors differ between browser and Node.js environments, I note the differences explicitly. Version-sensitive features are flagged with their introduction version so you can track compatibility.

Let us begin.

Chapter 1: WebAssembly Fundamentals for Pyodide

What Is WebAssembly and Why Does It Matter?

WebAssembly (commonly abbreviated as WASM) is a binary instruction format designed to run efficiently in modern web browsers and other environments. Think of it as a portable, low-level virtual machine that sits alongside JavaScript in the browser execution stack. Unlike JavaScript, which was designed as a scripting language for interactive web pages, WebAssembly was engineered from the ground up for high-performance execution of compiled code.

The key insight behind WebAssembly is simple: many applications need performance beyond what interpreted or just-in-time-compiled JavaScript can reliably provide. Games, video editing tools, CAD software, cryptographic operations, scientific computing: these workloads benefit from predictable, near-native execution speeds. WebAssembly makes it possible to write such code in languages like C, C++, Rust, and now effectively Python (through Pyodide), compile it to a portable binary format, and run it in any modern browser without plugins or platform-specific builds.

For Pyodide specifically, WebAssembly matters because it is the execution substrate that makes running CPython in the browser feasible at all. The CPython interpreter is written primarily in C. To run it inside a browser sandbox, someone had to compile that C code into something the browser could execute efficiently. WebAssembly is that something.

How WebAssembly Runs in Browsers

WebAssembly modules are binary files with the `.wasm` extension. They contain a sequence of instructions for a stack-based virtual machine, along with metadata about memory requirements, imported functions, and exported symbols. When a browser loads a WebAssembly module, it goes through three phases:

Fetching. The browser downloads the `.wasm` file, typically via an HTTP request triggered by JavaScript code or an HTML `<script type="module">` tag. Because WebAssembly files are binary, they compress very well, often achieving 70–80 percent compression ratios with gzip or Brotli. This matters for Pyodide, whose runtime is several megabytes when uncompressed but much smaller on the wire.

Compiling. The browser validates and compiles the WebAssembly binary into native machine code optimized for the host CPU architecture. Modern browsers use ahead-of-time compilation strategies that produce highly optimized code comparable to what you would get from a native compiler targeting the same platform. This compilation happens before any WebAssembly code executes, which is why initial load time can be noticeable for large modules.

Instantiation. The browser creates a runtime instance of the compiled module, allocating its required memory and linking imported functions (typically provided by JavaScript) to exported functions that JavaScript can call. Once instantiated, calling into WebAssembly is extremely fast, often within single-digit microseconds per call.

Here is what loading a WebAssembly module looks like in modern JavaScript:

```
// Modern approach using WebAssembly.instantiateStreaming
async function loadWasm() {
  const response = await fetch("module.wasm");
  const result = await WebAssembly.instantiateStreaming(response, {
    // Import object: functions the WASM module expects from JavaScript
    env: {
      print: (message) => console.log(message),
    },
  });

  // Exported functions from the WASM module are now callable
  const { add } = result.instance.exports;
  console.log("2 + 3 =", add(2, 3));
}
```

For Pyodide, this process is abstracted away by its JavaScript loader (`pyodide.js` or `pyodide.mjs`), which handles fetching the WebAssembly module, providing all required imports, and exposing a convenient API for running Python code. But understanding that there is a compile-and-instantiate step underneath helps explain why Pyodide initialization takes time on first load.

Emscripten: The Bridge from CPython to WebAssembly

Emscripten is the compilation toolchain that makes Pyodide possible. Developed by Mozilla and now maintained as an open-source project, Emscripten allows you to compile C and C++ code to WebAssembly (and optionally JavaScript glue code). It includes a full compiler infrastructure built on LLVM, a system library implementation called emscripten-newlib, and extensive support for POSIX-like APIs that many C programs depend on.

The basic workflow looks like this:

1. You start with C or C++ source code (in Pyodide's case, the CPython interpreter plus standard library modules).
2. Emscripten's clang frontend compiles the source to LLVM IR (intermediate representation).
3. LLVM optimizes and generates WebAssembly bytecode from the IR.
4. Emscripten produces a JavaScript "glue" file that handles module loading, memory management, and API bindings between JavaScript and the compiled code.

For Pyodide specifically, the build process is more complex because it must:

- Compile the entire CPython interpreter with modifications to work in a WebAssembly environment.
- Apply patches to handle missing or modified system calls (no fork(), no pthreads, different networking model).
- Bundle Python standard library modules and third-party packages as compressed archives loaded into a virtual filesystem at runtime.
- Generate foreign function interface bindings that allow Python code to call JavaScript functions and vice versa.

The resulting Pyodide distribution consists of several key files:

- `pyodide.js` or `pyodide.mjs`: The main JavaScript loader and API layer.
- `pyodide.asm.wasm`: The compiled WebAssembly module containing the CPython runtime.
- `python_stdlib.zip`: A compressed archive of Python standard library modules.

- `pyodide-lock.json`: A lockfile describing available packages, their sizes, hashes, and dependencies.

These files are what get loaded when you initialize Pyodide in your application. The JavaScript loader fetches them from a CDN or local server, instantiates the WebAssembly module, initializes the Python runtime, and then exposes an API for running Python code.

WebAssembly Memory Model and Its Implications

Understanding WebAssembly's memory model is crucial for using Pyodide effectively, particularly when dealing with large data structures, performance optimization, and debugging memory-related issues.

WebAssembly uses a **linear memory model**: each module has a single contiguous block of memory that grows in fixed-size pages. Each page is exactly 64 kilobytes. When a WebAssembly module is instantiated, it starts with a specified initial number of pages and can optionally grow by allocating additional pages as needed (up to a maximum).

For Pyodide running on current browsers:

- The default initial memory is typically around 256 MB (4096 pages).
- Memory can grow dynamically as Python allocates objects.
- The practical limit for 32-bit WebAssembly (the current standard) is approximately 4 GB, though Pyodide's default configuration caps lower at around 2 GB to avoid exhausting browser memory budgets [13].

This linear memory model has several important implications:

Shared memory between Python and JavaScript. Because both the Python runtime and the host JavaScript environment share access to the same WebAssembly memory buffer, it is possible to implement zero-copy data transfers. When you pass a NumPy array from Python to JavaScript (or vice versa), Pyodide can provide a view onto the same underlying memory rather than copying all the data. This dramatically improves performance for large arrays but requires careful lifetime management to avoid accessing freed memory.

Memory cannot be shrunk. Once WebAssembly allocates additional pages, they cannot be released back to the browser. If your application loads and

processes large datasets temporarily, that memory stays allocated even after you delete the Python objects. This is a consideration for long-running applications that process variable-sized data.

No direct filesystem access. The linear memory model means WebAssembly modules cannot directly access the host operating system's file system. Instead, Emscripten provides virtual filesystem implementations (MEMFS, IDBFS, etc.) that run entirely within the WebAssembly memory space. Python file operations in Pyodide work with these virtual filesystems rather than real files on disk. We cover this in detail in Chapter 6.

Memory limits are hard constraints. Unlike JavaScript's garbage collector, which can reclaim memory from unused objects relatively freely, WebAssembly memory growth is one-way and bounded by the maximum page limit. If your Python code tries to allocate more memory than available, you get an out-of-memory error with no graceful degradation path. This makes memory profiling important for production Pyodide applications that handle large datasets.

Here is how you can check current memory usage from within Pyodide:

```
import sys

# Check total allocated memory (in bytes)
total_mem = sys.getsizeof(__builtins__)  # rough indicator only

# For more detailed info, inspect the WebAssembly memory from JavaScript side
# pyodide._module.HEAPU8.byteLength gives current memory size in bytes
```

A more accurate approach is to check from JavaScript:

```
// After Pyodide is loaded
const memorySize = pyodide._module.HEAPU8.byteLength;
console.log(`Current WebAssembly memory: ${(memorySize / 1024 /
↪  1024).toFixed(2)} MB`);
```

Performance Characteristics of WebAssembly

One of the most common questions about Pyodide is: how fast is it compared to native Python? The answer depends on what you are measuring, but here is a realistic picture based on benchmarks and real-world usage.

Startup time. This is Pyodide's biggest performance challenge. Loading and initializing Pyodide for the first time typically takes 2–5 seconds depending on network conditions, because the browser must download several megabytes of code (the WebAssembly module, standard library archive, and JavaScript loader) and compile the WebAssembly into native machine code. On subsequent visits, if the files are cached by the browser, initialization drops to roughly 0.5–1 second. This is acceptable for interactive tools but problematic for applications where users expect instant responsiveness.

Execution speed. For pure Python code (code without C extensions), Pyodide runs at roughly 60–80 percent of native CPython performance on comparable hardware [14]. The overhead comes from the WebAssembly execution layer and the fact that some operations that are cheap in native code require additional indirection in WebAssembly. For compute-intensive workloads using optimized C extensions (NumPy linear algebra, pandas groupby operations), performance is closer to 80–95 percent of native because those hot paths run as compiled WebAssembly code rather than interpreted Python bytecode.

Memory allocation. Allocating and garbage collecting objects in Pyodide is slower than in native CPython, typically by a factor of 1.5–2x. This matters most for applications that create many short-lived objects (e.g., parsing large text files line by line, processing millions of small records).

JavaScript interop overhead. Every time you call from Python to JavaScript or vice versa, there is overhead for marshaling arguments and return values across the language boundary. Simple calls add microseconds of latency; passing large data structures can add milliseconds. For tight loops that cross the boundary frequently, this overhead becomes significant. The general rule is: minimize cross-language calls in performance-critical paths. Batch operations when possible.

Browser constraints. Remember that Pyodide runs inside a browser tab, which means it shares resources with everything else the user has open. Browser memory budgets per tab are limited (often 1–2 GB on desktop browsers, less on mobile). Heavy computation can make the UI unresponsive unless you run Pyodide in a Web Worker (covered in Chapter 11). Battery life on mobile devices is also affected by intensive computation.

Despite these limitations, Pyodide's performance is good enough for many real-world use cases:

- Interactive data exploration with datasets up to tens of millions of rows.
- Client-side machine learning inference with pre-trained models.
- Real-time signal processing and audio analysis.
- Educational tools where correctness and accessibility matter more than raw speed.
- Edge computing functions where Python's expressiveness is worth the startup overhead.

The key is to design your application with Pyodide's performance profile in mind from the start, rather than discovering limitations after building something that assumes native-speed execution. Later chapters cover specific optimization strategies for different workload types.

Summary

WebAssembly provides the execution substrate that makes running CPython in browsers possible. Emscripten compiles CPython's C code to WebAssembly, and Pyodide wraps this with Python package management, JavaScript interoperability, and browser integration. Understanding the underlying memory model, compilation process, and performance characteristics helps you make better architectural decisions and avoid common pitfalls.

In the next chapter, we move from theory to practice: you will set up Pyodide in multiple environments and run your first Python code in the browser.

Chapter 2: Getting Started with Pyodide

Your First Pyodide Page via CDN

The fastest way to try Pyodide is to load it from a Content Delivery Network (CDN) directly in an HTML file. No installation, no build tools, no configuration: just create an HTML file and open it in your browser.

JsDelivr is the primary CDN hosting Pyodide distributions. The URL pattern follows this structure:

```
https://cdn.jsdelivr.net/pyodide/v{version}/full/pyodide.js
```

Replace `{version}` with a specific release like `v314.0.4` for stability, or use `latest` for the cutting edge (not recommended for production).

Here is a complete, minimal HTML file that loads Pyodide and runs Python code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello Pyodide</title>
</head>
<body>
    <h1>Hello from Python in the Browser</h1>
    <div id="output"></div>

    <!-- Load Pyodide from CDN -->
    <script type="module">
        import { loadPyodide } from
        ↪  "https://cdn.jsdelivr.net/pyodide/v314.0.4/full/pyodide.js";

        async function main() {
            // Show loading indicator
```

```
            const output = document.getElementById("output");
            output.textContent = "Loading Pyodide...";

            // Initialize Pyodide
            let pyodide = await loadPyodide();

            // Run Python code and display the result
            let result = pyodide.runPython(`
                def greet(name):
                    return f"Hello from Python 3.14, {name}!"

                greet("World")
            `);

            output.textContent = result;
        }

        main().catch(err => {
            document.getElementById("output").textContent = "Error: " +
            ↪  err.message;
        });
    </script>
</body>
</html>
```

Save this as `hello-pyodide.html` and open it in your browser. You should see "Loading Pyodide..." briefly, followed by "Hello from Python 3.14, World!" after a few seconds.

Let us break down what is happening:

- **`<script type="module">`**: We use ES modules syntax because `loadPyodide` is exported as an ES module. This allows us to use `import` directly in the browser without a bundler.
- **`import { loadPyodide } from "..."`**: Imports the Pyodide loader function from the CDN URL. The URL points to version 314.0.4's full distribution; the `full/` path includes all standard packages bundled with that release.
- **`await loadPyodide()`**: This is where the heavy lifting happens. The function fetches the WebAssembly module (`pyodide.asm.wasm`), the compressed Python standard library (`python_stdlib.zip`), and other required files from the CDN, then instantiates the WebAssembly module and initializes the Python runtime. This is why you see a delay on first load.

- **`pyodide.runPython(...)`**: Executes the Python code string inside the initialized interpreter. The code defines a function `greet` and calls it. The return value of the last expression in the code string becomes the JavaScript return value, in this case a Python string converted to a JavaScript string.
- **Error handling**: We wrap everything in an async function with `.catch()` to handle initialization or execution errors gracefully rather than leaving the user staring at "Loading Pyodide..." forever.

Important note about version pinning: In production applications, always pin to a specific version number (like `v314.0.4`) rather than using `latest` or unversioned URLs. Using `latest` means your application could break unexpectedly when Pyodide releases a new version with API changes. The development documentation explicitly warns against using non-versioned dev URLs for deployed applications [15].

Interactive REPL Example

A more practical first example is an interactive Python REPL (read-eval-print loop) where users can type and execute Python code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Pyodide REPL</title>
    <style>
        body { font-family: monospace; margin: 20px; }
        #output {
            background: #1e1e1e;
            color: #d4d4d4;
            padding: 10px;
            height: 300px;
            overflow-y: auto;
            white-space: pre-wrap;
            border-radius: 4px;
        }
        #code { width: 100%; min-height: 60px; font-family: monospace; padding:
        ↪  8px; }
        button { margin-top: 8px; padding: 8px 16px; cursor: pointer; }
    </style>
</head>
```

```
<body>
    <h2>Pyodide REPL</h2>
    <textarea id="code" placeholder="Type Python code here...">print("Hello
    ↪  from Pyodide!")
import sys
print(f"Python version: {sys.version}")</textarea>
    <br>
    <button id="run">Run</button>
    <div id="output"></div>

    <script type="module">
        import { loadPyodide } from
        ↪  "https://cdn.jsdelivr.net/pyodide/v314.0.4/full/pyodide.js";

        const output = document.getElementById("output");
        const code = document.getElementById("code");
        const runButton = document.getElementById("run");

        function addToOutput(text) {
            output.textContent += text + "\n";
            output.scrollTop = output.scrollHeight;
        }

        // Initialize Pyodide
        output.textContent = "Initializing Pyodide...";
        runButton.disabled = true;

        async function main() {
            let pyodide = await loadPyodide();
            addToOutput("Ready!\n");
            runButton.disabled = false;

            // Store pyodide globally so the button handler can access it
            window.pyodide = pyodide;
        }

        main().catch(err => {
            addToOutput("Initialization error: " + err.message);
        });

        // Handle Run button clicks
        runButton.addEventListener("click", async () => {
            const pyodide = window.pyodide;
            if (!pyodide) return;

            addToOutput(">>> " + code.value.replace(/\n/g, "\n>>> ") + "\n");

            try {
                let result = pyodide.runPython(code.value);
```

```
                // Only show result if it's not None (runPython returns
                ↪ undefined for None)
                if (result !== undefined) {
                    addToOutput(String(result));
                }
            } catch (err) {
                addToOutput("Error: " + err);
            }

            code.value = "";
        });
    </script>
</body>
</html>
```

This example demonstrates several patterns you will encounter repeatedly:

- **Global state management**: We store the `pyodide` instance on `window` so it persists across button clicks. In larger applications, you would manage this state more carefully (perhaps in a dedicated module or using framework state management).
- **Capturing print output**: Notice that `print()` statements inside Python code automatically appear in our output div. This happens because Pyodide's default stdout/stderr handlers write to the JavaScript console by default. To capture them in your UI, you would override these handlers (covered in Chapter 8).
- **Error handling at execution time**: Python errors are caught as JavaScript exceptions with descriptive messages that include traceback information. Users can see what went wrong and fix their code.

Installing Pyodide with npm for Modern Projects

While CDN loading works great for simple pages and prototypes, modern web development typically uses package managers and build tools. Pyodide is available on npm as the `pyodide` package, making it easy to integrate into projects using Vite, Webpack, or similar tooling.

First, create a new project directory and initialize npm:

```
mkdir my-pyodide-app
cd my-pyodide-app
npm init -y
npm install pyodide
```

The npm package includes the Pyodide JavaScript loader and core runtime files, but it does not include the full set of Python packages (those would add hundreds of megabytes). Instead, you configure `loadPyodide` to fetch packages from a CDN or self-hosted location at runtime.

Here is a simple Vite-based project structure:

```
my-pyodide-app/
├── index.html
├── package.json
├── src/
│   ├── main.js
│   └── style.css
└── vite.config.js
```

`index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Pyodide + Vite</title>
</head>
<body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
</body>
</html>
```

`src/main.js`:

```
import { loadPyodide } from "pyodide";
import "./style.css";

const app = document.getElementById("app");

async function init() {
    app.textContent = "Loading Pyodide...";

    // Configure Pyodide to load packages from CDN
    const pyodide = await loadPyodide({
        indexURL: "https://cdn.jsdelivr.net/pyodide/v314.0.4/full/",
    });

    // Run Python code
    const result = pyodide.runPython(`
        2 + 2
    `);

    app.innerHTML = `
        <h1>Pyodide + Vite</h1>
        <p>Python says 2 + 2 = ${result}</p>
        <p>Running in browser with Python ${pyodide.runPython("import sys;
↪   sys.version")}</p>
    `;

    // Keep pyodide available for later use
    window.pyodide = pyodide;
}

init().catch(err => {
    app.textContent = "Error: " + err.message;
    console.error(err);
});
```

`vite.config.js`:

```
import { defineConfig } from "vite";

export default defineConfig({
    // Exclude pyodide from Vite's dependency pre-bundling
    // because it contains WebAssembly and special file handling
    optimizeDeps: {
        exclude: ["pyodide"],
    },
});
```

Run the development server with `npx vite`, then open `http://localhost:5173` in your browser. You should see Pyodide load and display the result of a Python

computation.

Why exclude pyodide from optimizeDeps? Vite's dependency pre-bundling uses Rollup to bundle dependencies for faster development loading. However, Pyodide's module structure includes WebAssembly files and special exports that do not bundle cleanly with default settings. Excluding it ensures the loader works correctly during development.

For production builds where you want to self-host Pyodide files rather than rely on a CDN, you need additional configuration to copy Pyodide's runtime files into your build output. We cover bundler integration in detail in Chapter 14.

Running Pyodide in Node.js

Pyodide can also run in Node.js environments, which is useful for testing Python code locally without a browser, building CLI tools that embed Python, or running Pyodide-based logic on serverless platforms. Starting with Pyodide version 0.25.0, Node.js 18 or later is required [16].

Install Pyodide in your Node.js project:

```
npm install pyodide
```

Create a file called `hello-node.js`:

```
import { loadPyodide } from "pyodide";

async function main() {
    // When running in Node.js, indexURL should be empty or point to local files
    // The npm package includes the core runtime files
    const pyodide = await loadPyodide({
        indexURL: "",
    });

    console.log("Pyodide loaded in Node.js");

    // Run Python code
    const result = pyodide.runPython(`
        import sys
        print(f"Running Python {sys.version}")
        "Hello from Pyodide in Node.js!"
```

```
    `);

    console.log("Result:", result);

    // Install a package using micropip
    await pyodide.runPythonAsync(`
        import micropip
        await micropip.install("requests")
        import requests
        print(f"Requests version: {requests.__version__}")
    `);
}

main().catch(console.error);
```

Run it with:

```
node hello-node.js
```

You should see Python version information printed, followed by the result string and the installed requests library version.

Key differences when running in Node.js:

- **Filesystem access**: In Node.js, you can mount the host filesystem using NODEFS (covered in Chapter 6), giving your Python code access to real files on disk. This is not available in browsers.
- **Network requests**: Node.js provides different networking APIs than browsers. Pyodide adapts its HTTP client accordingly, but some browser-specific CORS behaviors do not apply.
- **No DOM**: Obviously, there is no browser DOM in Node.js. Code that accesses `js.document` or other browser APIs will fail.
- **Package caching**: Browser environments cache downloaded packages automatically. In Node.js, package downloads are not cached by default unless you configure caching explicitly.

For interactive experimentation in Node.js, you can use the REPL with top-level await support:

```
node --experimental-repl-await
```

Then type:

```
> const { loadPyodide } = require("pyodide");
> let pyodide = await loadPyodide({ indexURL: "" });
> await pyodide.runPythonAsync("1 + 1");
2
```

Understanding the Initialization Process

When you call `loadPyodide()`, several things happen behind the scenes. Understanding this process helps with debugging initialization issues and optimizing load times.

Step 1: Fetch the JavaScript loader. The browser or Node.js downloads and executes `pyodide.js` (or `pyodide.mjs`). This file sets up the module loading infrastructure and defines the `loadPyodide` function.

Step 2: Load configuration. The loader reads the `indexURL` option you provided and uses it to construct URLs for other required files. If you did not specify `indexURL`, it defaults to a CDN URL based on the loaded script's origin.

Step 3: Fetch and compile WebAssembly. The loader downloads `pyodide.asm.wasm` (the compiled CPython runtime) and passes it to the browser's WebAssembly compiler. This is typically the slowest step on first load because the file is several megabytes and requires compilation to native machine code.

Step 4: Load the Python standard library. The loader downloads `python_stdlib.zip`, a compressed archive containing Python standard library modules, and extracts it into Pyodide's virtual filesystem (MEMFS by default). This makes standard library imports like `import os` or `import json` work immediately.

Step 5: Initialize the Python runtime. The WebAssembly module is instantiated with its memory buffer and imported functions. The CPython interpreter initializes itself, including setting up the GIL (global interpreter lock), importing core modules, and establishing the foreign function interface with JavaScript.

Step 6: Return the PyodideAPI object. Once initialization completes, `loadPyodide` resolves with a `PyodideAPI` object that provides methods for running Python code, loading packages, accessing the filesystem, and more.

The entire process typically takes 2–5 seconds on first load over a moderate-speed connection, with most time spent downloading and compiling the WebAssembly module. Subsequent loads are faster if files are cached by the browser.

You can monitor initialization progress by providing callback functions:

```
const pyodide = await loadPyodide({
    stdout: (message) => console.log("Python stdout:", message),
    stderr: (message) => console.error("Python stderr:", message),
    loadPackage: async (name, packageList, message) => {
        console.log(`Loading package: ${name}`);
    },
});
```

The `loadPackage` callback is particularly useful when loading additional packages after initialization, as it lets you show progress indicators to users.

Common First-Time Setup Pitfalls

Here are issues new Pyodide users frequently encounter and how to resolve them:

"loadPyodide is not defined": This usually means the script failed to load or you are using the wrong import syntax. If loading from CDN with a `<script>` tag (not module), use `window.loadPyodide`. With ES modules, ensure you use `<script type="module">` and proper import syntax.

CORS errors when loading from local file: Opening an HTML file directly via `file://` protocol often triggers CORS restrictions when Pyodide tries to fetch its companion files. Use a local development server instead: `python -m http.server` or `npx serve` in the directory containing your HTML file, then open `http://localhost:8000`.

"ModuleNotFoundError: No module named 'numpy'": Only the Python standard library is available immediately after loading Pyodide. Third-party packages like NumPy must be loaded explicitly using `pyodide.loadPackage("numpy")` from JavaScript or `await`

`micropip.install("numpy")` from Python. We cover package management in Chapter 5.

UI freezing during initialization: Pyodide's WebAssembly module runs on the main browser thread by default, which blocks UI updates during loading and heavy computation. For production applications, run Pyodide in a Web Worker to keep the interface responsive (covered in Chapter 11).

Version mismatches between CDN and npm package: If you install `pyodide` via npm but configure `indexURL` to point to a different version on the CDN, you may encounter compatibility issues. Ensure versions match, or use the npm package's bundled files with an empty `indexURL`.

Summary

You now have Pyodide running in three environments: a simple HTML page loaded from CDN, a modern Vite-based project using npm, and a Node.js script. You understand the initialization process and know how to troubleshoot common first-time issues.

In the next chapter, we dive deeper into Pyodide's architecture and runtime internals. Understanding how Pyodide works under the hood will make you a more effective developer when debugging issues, optimizing performance, and designing complex applications.

Chapter 3: Pyodide Architecture and Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

CPython Compiled to WebAssembly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The Pyodide Runtime Initialization Sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Memory Layout: WASM Heap and Python Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The Execution Model and Event Loop Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Versioning and the Road to CPython Tier 3 Status

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 4: JavaScript–Python Interoperability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The Foreign Function Interface Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Implicit Type Conversions Between Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

JsProxy: Accessing JavaScript from Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

PyProxy: Passing Python Objects to JavaScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Calling Functions Across the Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Memory Management and Preventing Leaks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 5: Package Management in Pyodide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

What Comes Built Into Pyodide?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Loading Official Packages with loadPackage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Using micropip for PyPI Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Custom Wheel URLs and CORS Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Dependency Resolution and Optimization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 6: Filesystems and I/O Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The Virtual Filesystem Concept

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

MEMFS: In-Memory Files by Default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Persistent Storage with IDBFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Mounting Other Filesystems (NODEFS, PROXYFS)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Working with Binary Data and Buffers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 7: Asynchronous Programming Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Asyncio in a Single-Threaded Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The WebLoop Event Loop Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

runPythonAsync and Top-Level Await

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Coordinating Python Coroutines with JavaScript Promises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

JavaScript Promise Integration (JSPI) Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 8: Browser APIs and Web Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Accessing the DOM via the js Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Canvas Rendering and Graphics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Handling Browser Events from Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

LocalStorage and SessionStorage Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Custom JavaScript Modules for Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 9: Data Handling and Visualization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Loading the Scientific Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Working with NumPy Arrays Efficiently

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

DataFrame Operations with Pandas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Rendering Matplotlib Plots in the Browser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Zero-Copy Data Transfer Between Python and JavaScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 10: Networking, HTTP, and Serverless Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Making HTTP Requests with pyfetch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Handling CORS and Browser Security Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Synchronous vs Asynchronous HTTP Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Running Python on the Edge with Cloudflare Workers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Designing Serverless Architectures with Pyodide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 11: Performance Optimization Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Understanding Pyodide's Performance Profile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Reducing Initial Load Time and Bundle Size

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Memory Management Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Using Web Workers to Avoid UI Blocking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Optimizing Data Transfer Across the Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 12: Debugging and Development Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Console Logging and Basic Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Understanding Python Errors in the Browser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Debugging JavaScript–Python Boundary Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Advanced WebAssembly Debugging Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Development Tools and Build Configurations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 13: Security Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The WebAssembly Sandbox Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Content Security Policy Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Trust Boundaries Between JavaScript and Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Running Untrusted Code Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Common Security Pitfalls and Mitigations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 14: Packaging, Deployment, and Production Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Self-Hosting vs CDN Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Working with Bundlers (Vite, Webpack, etc.)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Integration with React, Vue, and Svelte

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Building Custom Pyodide Packages with pyodide-build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Production Monitoring and Error Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Chapter 15: Limitations, Trade-offs, and the Future

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Unsupported Standard Library Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Threading, Multiprocessing, and Concurrency Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Memory Constraints and Large Data Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

When Not to Use Pyodide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

The Road Ahead: CPython Integration and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pyodideinpractice.